CHIMERA II: A Real-Time UNIX-Compatible Multiprocessor Operating System for Sensor-based Control Applications

David B. Stewart
Donald E. Schmitz
Pradeep K. Khosla

CMU-RI-TR-89-24

Contents

List of Figures

Abstract

This paper describes the CHIMERA II multiprocessing operating system, which has been developed to provide the flexibility, performance, and UNIX-compatible interface needed for fast development and implementation of parallel real-time control code. The operating system is intended for sensor-based control applications such as robotics, process control, and manufacturing. The features of CHIMERA II include support for multiple general purpose CPUs; support for multiple special purpose processors and I/O devices; a high-performance real-time multitasking kernel; user redefinable dynamic real-time schedulers; a UNIX-like environment, which supports most standard C system and library calls; standardized interrupt and exception handlers; and a user interface which serves to download, monitor, and debug code on any processor board, and serves as a terminal interface to the executing code. CHIMERA II also offers an attractive set of interprocessor communication features. The system-level *express mail* facility provides transparent access to a host file system and remote devices, and provides the basis for implementing user-level interprocessor communication. Application programmers have the choice of using shared memory, message passing, remote semaphores, or other special synchronization primitives for communicating between multiple processors. As an example of an actual implementation, we are currently using CHIMERA II to control a multi-sensor based robot system. The system runs on a Sun workstation host, with one or more Ironics M68020 processing boards, connected over a VME backplane. The system contains various special purpose processors, including a Mercury 3200 Floating Point Unit and an Androx Image Processor. The system also supports a variety of sensors and devices for real-time systems, which currently include a camera, force and [illegible]e sensors, and a joystick.

Carnegie
Mellon

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

7 December 1989

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314

Attn: J. Cundiff

Dear Mr. Cundiff:

RE: Report No. CMU-RI-TR-89-24

Permission is granted to the Defense Technical Information Center and the National Technical Information Service to reproduce and sell the following report, which contains information general in nature:

> *CHIMERA II: A Real-time UNIX-Compatible Multiprocessor Operating System for Sensor-based Control Applications*, by David B. Stewart, Donald E. Schmitz, and Pradeep K. Khosla.

Yours truly,

Nancy A. Serviou

Nancy A. Serviou
Document Cordinator

enc.: 12 copies of report

1 Introduction

Sensor-based control applications, such as robotics, process control, and manufacturing systems, present problems to conventional operating systems because of their need for several different hierarchical levels of control, which can fall into three categories: *servo levels*, *supervisory levels*, and *planning levels*. The **servo levels** involve reading data from sensors, analyzing the data, and controlling electro-mechanical devices, such as robots or machines. The timing of these levels is critical, and often involves periodic processes ranging from 10 Hz to 1000 Hz. The **supervisory levels** are higher level actions, such as specifying a task, issuing commands like *turn on motor 3* or *move to position B*, and selecting different modes of control based on data received from sensors at the servo level. Time at this level is a factor, but not as critical as for the *servo* levels. In the **planning levels** time is not a factor. Examples of processes in this level include generating accounting or performance logs of the real-time system, simulations, and programming new tasks for the system to take on.

In order to satisfy the needs of sensor-based control applications, a flexible real-time, multitasking and parallel programming environment is needed. For the servo levels, it must provide a high performance real-time kernel, low-level communication, fast context switching and interrupt latency times, and support for special purpose CPUs and I/O devices. For the supervisory levels, a message passing mechanism, access to a file system, and scheduling flexibility is desired. Finally, the real-time environment must be compatible with a host workstation, which can provide tools for programming, debugging, and off-line analysis, are required by the planning levels. Ideally, a popular high level language is available to minimize the learning time of the system. The real-time operating system should also be designed so that programs running in simulation under a time-sharing environment can be incorporated into the real-time environment with minimal effort. CHIMERA II provides such an environment that is capable of supporting all levels of sensor-based control applications on a parallel computer system.

Several real-time operating systems currently exist for control type applications, such as VRTX, by Ready Systems [1], and VxWorks, by Wind River Systems [2]. VRTX is aimed at embedded systems for production, as opposed to CHIMERA II, which is a flexible UNIX-compatible real-time environment, suitable for research and development of control applications. VxWorks is similar to CHIMERA II in concept, but it is geared towards networking multiple single board computers to provide parallel processing power. It does not offer the necessary features for taking advantage of multiple processors on a common backplane, which is especially needed to satisfy the high compu-

tational demands of the servo levels. It also does not provide the scheduling and communication flexibility needed by many sensor-based control applications. In contrast, CHIMERA II is designed especially for providing maximum performance and flexibility in a parallel processing system. It takes advantage of shared memory over a backplane to reduce overhead on message passing and to provide interprocessor shared data segments and semaphores. The design of the CHIMERA II operating system, and its predecessor CHIMERA[3], was influenced by its target application, that involved real-time control of the Reconfigurable Modulator Manipulator System [4] and Direct Drive Arm II [5] at Carnegie Mellon University. The features, high performance, and flexibility of CHIMERA II allow it to be used in any type of process control, manufacturing, or real-time control applications; and as a testbed for research in real-time systems.

The remainder of this paper describes the implementation of CHIMERA II, and concentrates on the details of the operating system which make it unique. Section 2 provides an overview of the features of CHIMERA II. Section 3 describes the possible hardware that can be used with CHIMERA II. Section 4 provides details of the CHIMERA II real-time kernel, including our approach for obtaining low context switching times, scheduling flexibility, and real-time process control primitives. Section 5 describes the various forms of system-level and user-level interprocessor communication within the system.

2 CHIMERA II Features

CHIMERA II provides the necessary features for implementing sensor based control applications in a parallel computing environment. These features include the following:

- Support for multiple general purpose CPUs;
- Support for multiple special purpose CPUs and I/O devices;
- A real-time multitasking kernel;
- User definable and dynamically selectable real-time schedulers;
- Transparent access to a host file system and to remote devices;
- Generalized and efficient interprocess and interboard communication;
- Local and remote process synchronization;

- Standardized interrupt and exception handlers;
- UNIX-like environment, which supports most standard C system and library calls;
- Support for Hierarchical and Horizontal Control Architectures
- A user interface which serves to download, monitor, and debug code on any processor board, and serves as a terminal interface to the executing code.

One of the goals of CHIMERA II is to provide an environment for controlling systems that accept inputs for their operation from multiple sources that include both sensors and humans. Another goal is to develop an environment, based on commercially available devices, that can be easily ported and thus made widely available for research, development, and application. Based on these objectives, we established the following requirements for the hardware architecture of a programming environment for sensor-based control applications:

- Aside from application specific I/O, the hardware must be based on commercially available items.
- A well supported family of general purpose CPUs must be used in the entire system. These must be chosen for overall performance and software portability.
- A time sharing workstation must be used as a host to the real-time programming environment, in order to provide widely-used editors, debuggers, and window managers for program development.
- The real-time system must be expandable by adding one or more general purpose CPUs operating in parallel, each capable of working either independently or synchronized with other CPUs.
- The hardware must be capable of supporting special purpose devices, such as floating point processors and I/O devices, in order to provide enough flexibility for use in a wide variety of control applications.

Based on the above requirements, we chose to base CHIMERA II on the Sun 3 workstation, a popular and well supported Motorola M68020-based workstation with a VME bus. This choice dictated that our real-time engines also be VME-based M68020-based boards. Of the commercially available M68020 processor boards, we chose the Ironics family of CPU boards, because of support

for a local bus and *mailbox* interrupts, each of which helps in reducing potential memory bandwidth problems on the VME bus.

Besides the above requirements for the hardware architecture, we also established the following software requirements for a sensor-based real-time computing environment:

- The software environment must appear to the user as a real-time extension of a typical UNIX development system:
  - The C programming language must be available for *all* levels of the control program. There should not be any need for using assembly language.
  - The real-time kernel must support programs which are designed as multiple, concurrent processes. The kernel must support access to hardware devices via a library of high level routines, hiding the hardware details from the applications programmer.
  - Standard UNIX utility libraries must be ported or emulated, allowing ready portability of existing UNIX programs.
  - The UNIX file system must be accessible to all general processing boards.
- The kernel must be flexible enough to use a variety of schedulers to provide the best performance for a given task, thus taking advantage of the many different scheduling strategies which have been proposed for real-time operating systems.
- The software must implement a form of interprocessor communication and synchronization which requires a minimal amount of overhead, yet is flexible enough for all applications. Efficiency is very important since increased parallelism usually creates additional communication overhead between processors, which may nullify the advantages gained from making the code run in parallel.
- A standardized interface for interrupt handling and device drivers must be available, to simplify user code and also decrease development time of applications.
- The environment must provide the basic constructs required to support both hierarchical and horizontal control architectures, such as high level constructs for low-volume hierarchical communication; and low overhead communication for high-volume horizontal communication; and global shared memory across all processors.
- The simulation and real-time control environments must be similar, so that code can easily be moved between the two environments.

The above requirements must all be achieved by sacrificing a minimal amount of performance. The CHIMERA II programming environment provides a UNIX-compatible interface, which supports a real-time kernel, interprocessor communication, transparent access to the host file system and devices on the VME bus, remote process synchronization, and many more features needed for sensor-based control. In the remainder of this paper, we describe the hardware and software architecture for our implementation.

3 CHIMERA II Hardware Support

CHIMERA II is capable of supporting multiple general purpose processors (GPUs), which provide the parallel computational power needed to support multiple sensors in a control application. In addition, special purpose processors, such as floating point units (FPUs), image processing units (IPUs), and digital signal processors (DSPs), can be incorporated into the system. CHIMERA II also allows devices, such as serial ports, parallel ports, and frame grabbers, to be added to the system with relative ease. Each of these devices are accessible by all CPUs in the system. The remainder of this section describes the interface developed for CHIMERA II to incorporate a large variety of processors and devices within the system.

The minimum configuration to run the CHIMERA II environment is a Sun 3 workstation with VME backplane, running Sun OS 3.x or Sun OS 4.0.3[1], and one Ironics M68020 processing board. Figure 1 shows a sample hardware configuration. It is the one currently in use with the CMU Direct Drive Arm II project. The system consists of several processors and I/O devices:

- A Sun 3/260 host system on a VME bus, running Sun OS 4.0.3;
- A VME-to-VME bus adapter, to isolate the timesharing host from the real-time system;
- Multiple Ironics M68020 boards, possibly with different options, such as varying memory size, I/O ports, and local buses;
- A Mercury 3200 Floating Point Unit, which provides a peak performance of 20 Mflop, for intensive real-time control calculations;
- six Texas Instrument TMS320 DSP processors, on a Multibus backplane, each controlling one joint of the CMU DDArm II. The Multibus is connected to the VME bus through a

[1] We have plans to port the CHIMERA II code to the Mach operating system [6].

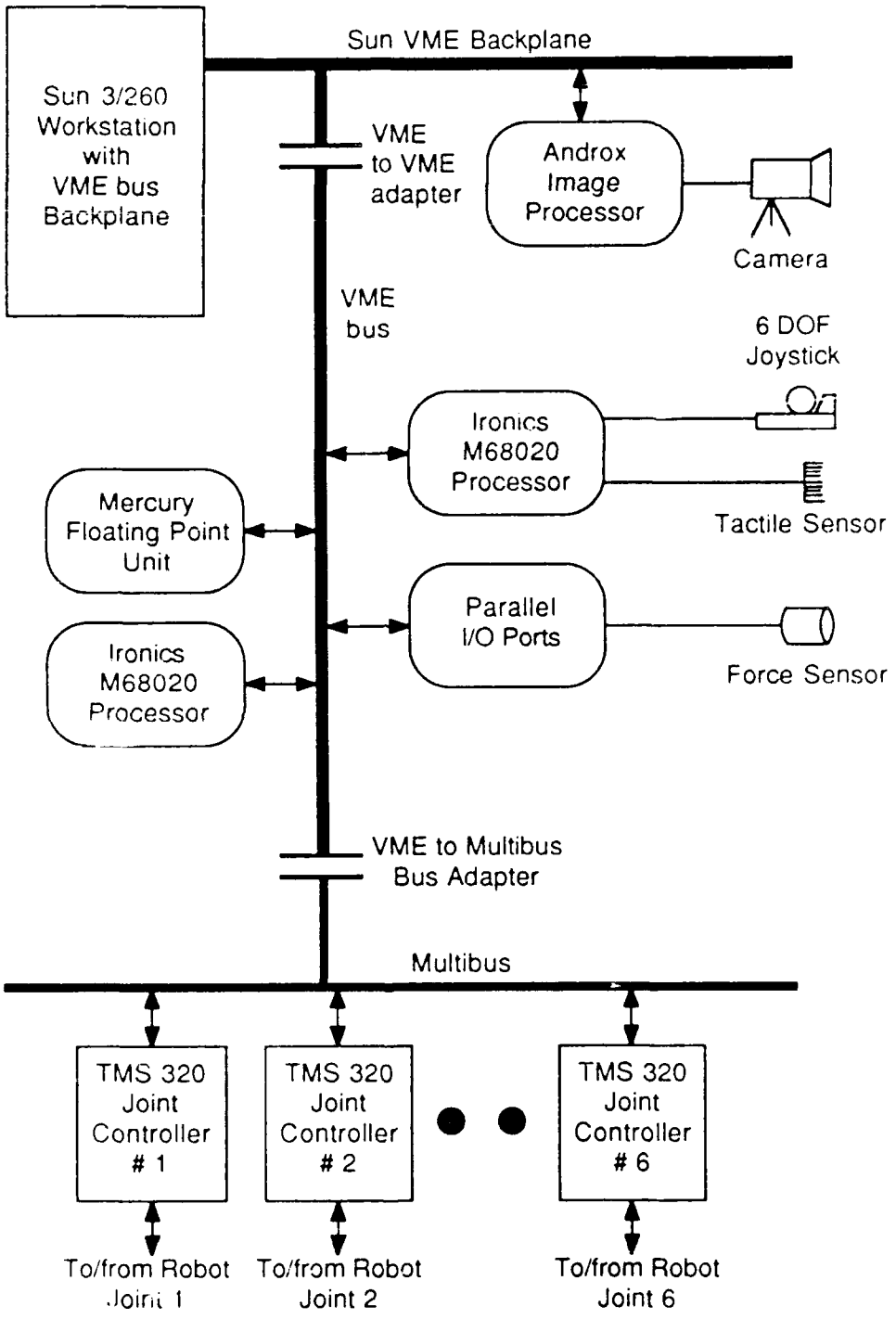

Figure 1. A Sample Hardware Configuration Supported by CHIMERA II

VME-to-Multibus Adapter.

- Multiple Sensors with corresponding I/O ports: a tactile sensor, connected to an Ironics serial port; a six axes force sensor, connected to a parallel I/O board; and a camera, with monitor, connected to an Androx Image Processing board;
- A six degree of freedom joystick, connected to an Ironics serial port.

The CHIMERA II kernel executes on each Ironics M68020 board, while the development environment is on the Sun host system. Non-68020 boards, such as the Mercury and the Androx, operate as slaves to one of the Ironics CPUs. Additional devices can be easily incorporated into the system.

Each Ironics M68020 has either an M68881 or M68882 floating point coprocessor on the board. Although these are useful for small amounts of floating point calculations, they are not powerful enough to support the computational requirements of a sensor-based control application. We opted for the Mercury 3200 floating point unit, since it offers the required performance at a reasonable cost; and because a true C compiler is supplied, allowing us to develop all parts of the control code in C.

The various sensors we are using not only allow us to perform research on multi-sensor control algorithms, but also provide a base for testing CHIMERA II's capability for handling a variety of devices. Similarly, the Mercury FPU, the Androx vision processor, and the custom TMS320 DSPs also provide a good way of testing CHIMERA II with a variety of special-purpose processors, hence demonstrating the flexibility of the programming environment. Finally, the six degree of freedom joystick is a source of aperiodic human interaction into the control loop of various algorithms.

3.1 General Purpose Processors

All CHIMERA II real-time programs execute on one or more *real-time engines* that are single board computers co-existing on the VME bus of a SUN-3 workstation *host*. At present, CHIMERA II supports the Motorola M68020 processor architecture as implemented on several commercial products, and will be ported to the newer M68030 in the near future. While ports to radically different CPU architectures are now under consideration, the M680X0 family provides sufficient performance for many real-time control applications and continues to improve. Using the same CPU family in both the real-time engines and host processors eliminates many of the problems of porting code between the simulation and real-time environments.

Real-time programs are typically written in C, and compiled using the standard SUN C compiler. Once compiled, these programs are linked with the real-time kernel, interprocessor communication package, and common utilities, all of which are simply UNIX library files, again using the standard SUN linker. The resulting executable file is then downloaded directly into the real-time processor's memory via the host's VME bus. A user interface program on the SUN host performs this downloading and provides a simple terminal-like interface to the remote processor, both to user programs running on the processor and to the on-board monitor program for debugging purposes.

3.2 Special Purpose Processors and I/O Devices

CHIMERA II uses the UNIX philosophy of device drivers to incorporate special purpose processors and I/O devices into the system. The UNIX-compatible calls *open()*, *close()*, *read()*, *write()*, *mmap()* and *ioctl()* are used to access these devices, thus hiding the details of the devices from the applications programmer. However, compared to UNIX, writing device drivers for CHIMERA II is simpler. Since there is no interprocess security or virtual memory (refer to Section 4.1 for details), much of the overhead and complexity required to write a set of device drivers has been eliminated. Each device is allocated a major number and a minor number. The major number is used as an index to the proper driver routines for the device, and the minor number distinguishes between multiple devices sharing the same driver.

A configuration file on the host workstation stores the information defining the processors and I/O devices within the system. Figure 2 shows the configuration file corresponding to our setup for the CMU DDArm II. The first part of the file contains lines beginning with "x", which lists the general processors, and defines their corresponding **xm** devices (see Section 5.2.1 for more details). Lines beginning with "#" are comments. The second part of the file defines all the processors and devices on the system (lines starting with "d"). For compatibility with the UNIX file system, all filenames are under the directory */vme*. Any access to a device with such a name is intercepted by CHIMERA II, and assumed to be located within the real-time processing environment; otherwise, CHIMERA II assumes that it is a file on the Sun host file system.

The different fields for defining general processing boards, in the first part of the file, are the following:

- *bid*: Numerical ID given to processor board

- *boardname*: Symbolic name given to the board
- *sunname*: Device name used on Sun host to access board
- *space*: The VME address space in which the device is defined.
- *vmeaddr*: The VME address of the board
- *cputype*: The model of the board.
- *vec*: For hxm# devices: Interrupt vector; the 3 least significant bits of the vector also represent the level of the interrupt. For ixm# devices: Mailbox number for mailbox interrupts.

All the information needed by the CHIMERA II system about the general processor boards are included in the configuration file. Similarly, for special purpose processors and devices, the necessary information is included, on lines starting with an "x". The fields are the following:

- *boardname*: The name of board which owns and is responsible for maintaining the data structures required for reading and writing the device.
- *vmename*: The name to be used in application programs wanting to open the device.
- *space*: The VME address space in which the device is defined. 'LOCAL' means that the device is only accessible to the local CPU.
- *vmeaddr*: The VME address of the board (or local address for LOCAL devices)
- *size*: Number of bytes occupied by device (memory mapped devices only)
- *maj*: Major device number
- *min*: Minor device number
- *vec*: Interrupt Vector generated by device (or 0 if device does not interrupt). The 3 least significant bits of the vector represent the level of the interrupt.

Usually special purpose processors are memory mapped for maximum performance, while the I/O drivers are accessed using the generic *read()* and *write()* system calls or utilities.

```
##############################################################################
#
# CHIMERA II Configuration file for CMU DDArm II
#
# This file contains the information needed to specify the configuration of
# the hardware.  Any devices specified in this file must have their
# corresponding device drivers installed if they are to be accessed by the
# user program.
#
##############################################################################
# declarations for general processing boards
#
# All numerical values in hex

# bid  boardname   sun-name  space    vmeaddr     cputype    vec
x 0      host     /dev/hxm0  A24D32    000000     SUN3       53
x 1     sensor    /dev/ixm0  A32D32  40000000     IV3220_4   01
x 2     control   /dev/ixm1  A24D32    800000     IV3204     01

# declarations for other devices on VME board

# /vme/sio# are the onboard serial devices of the Ironics IV3220 model.
# /vme/pio# are parallel I/O ports
# /vme/vmx0 allows mapping the Ironics local VMX bus into D32 space.
# /vme/vmx1 allows mapping the Ironics local VMX bus into D16 space.
# /vme/tms0 allows memory mapping the space of the 6 TMS320 processors.
# /vme/tmr0 are the control registers for tms0.
# /vme/mc0 is the mercury 3200 board.

# boardname vme-name   space    vmeaddr    size    maj  min  vec
d  sensor  /vme/sio0   LOCAL   FC460000      0      0    0    83
d  sensor  /vme/sio1   LOCAL   FC460000      0      0    1    83
d  sensor  /vme/pio0   A16D16      C0C0      0      1    0    74
d  sensor  /vme/pio1   A16D16      C0C0      0      1    1    74
d  sensor  /vme/pio2   A16D16      C0C0      0      1    2    74
d  sensor  /vme/pio3   A16D16      C0C0      0      1    3    74
d  control /vme/vmx0   LOCAL   FC000000      0      2    0     0
d  control /vme/vmx1   LOCAL   FD000000      0      2    1     0
d  control /vme/mc0    A32D32  b0000000   200000    3    0    e4
d  control /vme/tms0   A24D16    3d0000   030000    4    0     0
d  control /vme/tmr0   A16D16      0050       20    5    0     0

# End of Configuration file.
```

Figure 2: CHIMERA II Configuration File

4 CHIMERA II Kernel

The CHIMERA II multi-tasking capabilities are provided by a real-time executive or *kernel*. The kernel design exploits the unique requirements of a real-time control environment to provide much of the functionality of a conventional operating system with a minimal performance overhead. User access to the kernel is via a small set of process control primitives implemented as C callable library routines. These primitives support an Ada-like process control methodology, and serve as building blocks for more complex constructs. The kernel itself is a C library linked into the user's code, easing software modifications and user extensions. While the kernel implementation is fairly conventional, the design trade-offs are unique to the target environment. The various system requirements and their impact on kernel design are not always obvious, and are worthy of discussion here.

4.1 Real-time Computing Requirements

The computing requirements for a real-time system are very different from those of a conventional operating system. These constraints affect both the computing hardware and kernel design of a real-time system. In particular:

- Interprocess security is not required. In general, all of the processes running on a given CPU (or set of CPUs) are written and invoked by a single user — it is reasonable to assume that these processes are designed to cooperate. This eliminates much of the overhead in performing system calls or their equivalents, since all processes can be assumed to have all privileges.
- Programs are rarely limited by memory size. Real-time processes tend to be short repetitive operations, implying a small number of instructions and small data sets. This allows such systems to forgo virtual memory, improving memory system performance and eliminating memory management overhead from the process context switch operation.
- Process scheduling must occur at a fast rate, and include the concept of physical time and execution deadline in the scheduling algorithm. Restating the justification above, real-time processes tend to be active for short periods and deadline critical, requiring a scheduling time quanta on the order of the shortest physical event being controlled. Typical scheduling rates are an order of magnitude (or more) higher than conventional time-sharing operating systems. Since execution priorities are a function of time, this also implies re-evaluating the relative priority of each process every time quanta.

- The number of processes is usually small. This is due to the performance limitation of the CPU, and the difficulty of designing concurrent algorithms. This small number allows computationally intensive scheduling algorithms to be used without introducing unacceptable context switch overhead.

- Consistent or deterministic performance is more important than average performance. Many systems use techniques such as hardware caching or data dependency optimizations to increase the average performance. These features, however, are undesirable in a real-time system where predictability is more important than obtaining higher performance most of the time, at the cost of lower performance some of the time.

The CHIMERA II kernel is designed around the above requirements, and sacrifices interprocess security and virtual memory to provide the predictability and high performance needed for all levels of sensor-based control applications.

4.2 CHIMERA II Kernel Design

The CHIMERA II kernel is readily divided into five major components:

- Context switch code, written in assembly language, that performs the low level mechanics of saving and restoring the CPU state required to initiate a context switch. This code is initiated by either a hardware timer interrupt or a user level trap.

- A process scheduler, written in C, that is called from within the low level context switch. The scheduler is responsible for maintaining the current global process state, and selecting the next process to swap into the CPU.

- A set of data structures which contain the CPU state of non-active processes and the scheduling status of all processes.

- User interface routines that manipulate the process data structures to (indirectly) control the operation of the process scheduler, and thus the execution of the process.

- Exception handler code, written in C or assembler, which service traps caused by interrupts and execution errors, such as division by zero, illegal memory access, and illegal instruction.

Each of these components are described in more detail in the following paragraphs.

4.2.1 Context Switch Operation

The low-level context switch operation is very much dependent on the CPU architecture. The current CHIMERA II kernel supports the Motorola MC68020 CPU architecture, and uses a straightforward multi-tasking implementation. As is typical of many modern CPUs, the MC68020 enters a *supervisor state* when processing an exception, automatically toggling the active stack pointer from *user stack pointer* to *supervisor stack pointer*, and enabling all privileged instructions [2]. Each CHIMERA II process owns a supervisor stack area that is used as storage space for the CPU state when the process is inactive (not currently executing). Immediately after entering the exception handler code, the current CPU state is saved in this area. The process scheduler is then called (as a subroutine), which employs some algorithm to select the next process to execute, and returns a pointer to the new process's saved context. The current supervisor stack pointer is then replaced with the returned value, the CPU is loaded with the state information stored there, and the exception processing is exited. This effectively restarts the process defined by that state, at the point at which it was previously interrupted.

4.2.2 Process Scheduler

The operation of the process scheduler is the most significant difference between a real-time kernel and a conventional operating system. In addition to a much greater emphasis on execution efficiency, the real-time scheduler must select which process to make active as a function of physical time and the execution deadlines of the process pool. At the same time, the algorithm should impose a minimum burden on the user programmer — if the user must specify exactly when each process should be swapped in and out of the CPU, little is gained in terms of programming efficiency or performance over an interrupt driven approach.

The CHIMERA II scheduler algorithm was developed in an *ad hoc* fashion, incorporating various standard algorithms with extensive experimentation and tuning to obtain the best performance for typical job mixes. Given the lack of security requirements in this application, CHIMERA II does provide hooks into the context switch code to allow the user to replace the standard schedulers with application specific algorithms.

[2] While tasks normally run at user level, nothing prevents a task from explicitly changing the processor status and executing at supervisor level. In general there is no reason to do this; catastrophic failures due to programmer error are more likely when running in supervisor mode, however nothing prevents a user process from changing status if needed.

An important realization in the design of the scheduler was that there are two distinct reasons for performing a context switch:

1. The expiration of a time quanta, indicating it is time to re-evaluate the time based scheduling criteria, or simply to give another concurrent, equal priority process a chance to execute.
2. A process cannot continue due to contention for a system resource, for example mutually exclusive access to a data set or I/O device. In this case, the process which must wait may actually have a preference as to which process is to execute next in the remainder of its previously allocated time quanta.

In the first case, the standard CHIMERA II scheduler divides processes into two classes, those with real-time deadlines and those without. Processes with deadlines always have priority over those without, in the general case that multiple deadlines exist at once a *minimum-laxity-first* algorithm is used. This algorithm selects the process which must start execution in the shortest period in order to meet its deadline - ties are settled using a *highest-priority-first* algorithm. If no deadlines exist, processes are scheduled using a *highest-priority-first* algorithm. By replacing previously active processes at the end of the process ready list, and starting the next process search from the beginning, a *round-robin* behavior is introduced that improves scheduler fairness among equal priority processes.

Research has shown that the *minimum-laxity-first* algorithm provides very good real-time scheduling efficiency, and is inexpensive to implement[7]. The algorithm is also fairly simple to use. An application programmer has only to assign a relative priority to each process, and when required, specify the execution deadline, an estimate of the number of time quanta required to execute, and an *emergency action* routine. If a process misses its deadline, the scheduler automatically calls the emergency action routine, which can perform such tasks as aborting or restarting the process, altering the process's priority, or sending a message to another part of the system.

In the case of a context switch due to resource contention, a different algorithm is used. Since only a fraction of a time quanta remains, it is best to choose the next process quickly in order to utilize as much of the remaining time quanta as possible. In addition, the current processor hardware does not allow the scheduler to know how much of the time quanta remains, making it impossible to schedule based on physical time. Finally, the process which blocks may actually have a preference as to which process to run next; for example, a good strategy would be to give the process being waited on the remainder of the quanta. CHIMERA II addresses this diverse set of possibilities

by providing a context switch primitive which takes a scheduler routine as an argument. In most cases, the scheduler is a *round-robin* algorithm which simply selects the first process in the ready list, regardless of priority. The resulting context switch typically executes faster than the more complex time-driven algorithm. It is possible to specify a user supplied scheduler routine which implements an arbitrary algorithm to select the next process.

Our scheduling methodology, sometimes called *preemptive scheduling* [7] (we prefer *dynamic scheduling*), is in contrast to *rate monotonic* scheduling, in which each process is defined at compilation time as having periodic deadlines and well defined execution timing. Rate monotonic systems are often more efficient, as a provably optimal CPU utilization schedule can be determined offline, virtually eliminating the need for process scheduling at runtime (during a context switch) [8]. However, such an approach imposes more responsibility on the application programmer, slowing down program development and increasing the chance for an error. The rate monotonic methodology is also limited for the case of processes that require variable amounts of CPU utilization, such as iterative algorithms, which converge in a variable number of iterations, or aperiodic processes, which are usually triggered by an interrupt.

4.2.3 Process Data Structures

The CHIMERA II scheduler maintains process state information in a number of linked lists of *Process Control Blocks*, or *PCBs*. A PCB is simply a memory segment containing the process's supervisor stack, which is used for processing exceptions and also holds its CPU state when inactive, and a small amount of process specific scheduling information, such as priority and execution deadline. Each list corresponds to a particular state a process can be in (since there can only be one active or executing process at a time, there is no need for a list for this state):

- Ready to run with a deadline.
- Ready to run.
- Paused (waiting) on a timer to expire.
- Waiting on a software signal.

This sorting by execution state eliminates the number of processes which must be checked at any time for a potential change of state. In addition, the user is free to create lists indicative of more

complex process states and explicitly manipulate these processes using standard process control primitives.

The process list implementation is doubly linked with a fixed guard or head node that can never be removed. This implementation makes adding and removing arbitrary elements from a list efficient, requiring only a few processor instructions. Searching through the list is similarly efficient. Utilizing a guard node list representation also improves execution efficiency, eliminating the need for handling the special empty list case. The guard node also imposes a distinct ordering on the lists that is useful in many search algorithms.

4.2.4 Process Control Primitives

CHIMERA II provides the application programmer with a small set of *process control primitives* that support the most general level of process execution control. The CHIMERA II design philosophy has been to make these primitives as simple, general, and efficient as possible, allowing the user to build more complex operations on top of these primitives while achieving acceptable performance. The relative simplicity of the primitives makes them portable across hardware environments and processor architectures. Since (aside from hardware specific utilities) these primitives serve as the basis for all other CHIMERA II supported utilities, their functional specification can be thought of as defining the CHIMERA II programming environment. Following is a brief description of the most important *process control primitives*:

- *spawn()*: create an instance of a process.
- *pause()*: suspend a process for a specified (physical time) duration. Accepts optional parameters to specify a subsequent execution deadline.
- *set_deadline()*: specify the minimum number of time quanta of CPU utilization required by a process in a specific period to meet its execution deadline.
- *block(), wakeup()*: low level interprocess signaling mechanism.
- *P(), V()*: classic countered semaphore interprocess synchronization mechanism.
- *clock()*: returns the current physical time maintained by a hardware timer.
- *splx()*: UNIX-like interface to hardware interrupt mask, allowing processes to initiate uninterruptable atomic code.

From the users viewpoint, each CHIMERA II process is a C subroutine that can be considered to execute on its own (virtual) CPU, concurrently with an arbitrary number of other processes. Independent processes are allowed to schedule their execution as a function of physical time using the *pause()*, *set_deadline()* and *clock()* primitives. For example, a call to the primitive *pause()* sets the calling process's restart, required quanta and deadline fields to the arguments passed to *pause()*, moves the calling process to the *paused* process list and arranges a context switch to another *ready to run* process.

The CHIMERA II kernel, and in particular the process scheduler, then arranges to time-share the CPU among each process to provide each with the specified number of processor cycles required to meet their execution deadlines. The *block()*, *wakeup()*, *P()* and *V()* utilities are used in the more complex case in which concurrent processes must interact. These constructs support the concept of one process waiting for another, either in a producer-consumer relationship or as competitors for a global resource. The CHIMERA II scheduling algorithms support these features while still enforcing real-time execution constraints.

While a few of the above routines are implemented as conventional trap driven system calls, the majority are simply subroutines. Access to the CPU interrupt mask and status register is provided by the *splx()* routine — using this utility it is possible to code C routines that do practically anything that previously required assembly language coding[3].

4.2.5 Exception and Interrupt Handling

CHIMERA II allows users to write programs that respond to exceptions and hardware interrupts. These exception handlers are written as C routines, which are then called from a small segment of assembly code that provides the low level interface to the processor. CHIMERA II includes a preprocessor macro which automatically generates the proper assembly patch code when a routine is declared as an exception handler. Interrupt handlers are typically included in the *device driver* software that provides an interface between the CHIMERA real-time kernel and the hardware device as a standard part of the CHIMERA II software release, however there are no restrictions to prevent new interrupt handlers from being installed. More useful is the ability to define handlers for software exceptions, such as division by zero or attempting to access a non-existent memory location. By default, such exceptions are handled by printing an error message on the system

[3]Since these routines are very much CPU dependent such code will likely be non-portable across varying architectures.

console and terminating the offending process; however CHIMERA allows user programs to define an exception handler for such conditions on either a per-processor or per-process basis. The later requires making a copy of the standard exception vector table and installing it (with the modified entries) in place of the default table the vector table for the specified process.

4.3 Memory Management

Since no interprocess security is available in CHIMERA II, any process is capable of accessing any part of the processor's memory. However, unless there is some form of memory management, processes will not be able to use the memory efficiently. CHIMERA II provides the standard C routines *malloc(), free(), realloc(), etc.*, and other useful routines such as *mavail(), maxavail(), malloc_verify(), etc.* They provide the same functionality as their UNIX counterparts, but the implementation varies.

The routines keep two singly-linked lists: one for free blocks and one for allocated blocks. A first-fit algorithm is used to allocate blocks. Each block has a header, which includes the size of the allocated block, a pointer to the next block, and the process ID of the owner. Storing the owner's ID allows all allocated memory to be released when a process terminates, even if the process does not explicitly *free* the memory. The kernel on each board is responsible for it's own memory. Interprocessor shared memory segments are used to access off-board memory (see Section 5.3).

5 Interprocessor Communication

One of the most important aspects of a multiprocessor real-time system is its ability to provide fast, reliable, and standardized communication between all processes in the parallel computing environment. We have designed a standardized interface for low-overhead reliable communication among processes on parallel CPUs on a common backplane, including the Sun host. Multiple Sun hosts can then be connected to each other, by ethernet, using Sun's networking protocols, such as TCP/IP [9], RPC, and Sun NFS [10].[4] The features provided include shared memory, semaphores, and message passing, which not only work transparently across multiple processors, but also across the Sun host, allowing non-real-time processes to communicate with the real-time environment

[4]Note that since neither the ethernet nor UNIX operate in real-time, the concept of time is lost, and thus the real-time arena is exited. Programs that require such networking, however, are usually at the planning level, and thus do not need the fast real-time responses required for the servo and supervisory levels.

without the need of high-overhead networking protocols. The remainder of this section describes the communication facilities provided by CHIMERA II.

5.1 Global Shared Memory

The VME bus provides global shared memory by mapping the memory of each processor into one of several standard address spaces, depending on the addressing and data handling capabilities of the processor and memory. This form of memory mapping has the advantage that it provides the fastest possible communication with very little, if any, overhead. In such a scheme, however, several problems arise:

- Not all boards use the same address space. The VME bus alone supports several different modes (A32D32, A24D16, A16D16, etc.). Each board also views its own memory as a local address space, starting at address 0.
- The SUN host operates in a virtual memory environment, and although the SUN can easily map other CPU's memory into its virtual memory space, the reverse is not easily done.
- All information copied from one processor to another is untyped. The receiving board may not know what to do with the data unless it knows what data is arriving.
- Some mechanism is needed for different processors to settle on a single memory segment to communicate in. By default, two processes on different boards will not be able to communicate unless an absolute memory area is defined at compile time. This is not always desirable, nor always possible.
- Standard semaphores cannot be used for mutual exclusion. The kernel is usually responsible for controlling access to semaphores by blocking and waking up processes. In a multiprocessor environment, however, a kernel on one processor does not have control of blocking or waking up processes on another processor.

CHIMERA II solves the above mentioned problems, while still maintaining general and fast communication among all processors, by splitting the communication into two levels: system level and user level. The *system-level communication*, which we also call *express mail*, uses a combination of message passing and global shared memory. It is used only by the real-time kernels for performing remote operations transparent to the user and to communicate with the Sun Host; the user cannot

access these routines directly. The *user-level communication* is a set of high level primitives available to the applications programmer, which includes shared memory, message passing, and remote semaphores that can be used transparently across multiple processors.

5.2 System-Level Communication

In CHIMERA II, many UNIX system calls have been emulated as C procedures. Whenever these calls have to access a remote processor, a message is sent to the remote processor's *eXpress Mail* (xm) device. Each board has one **xm** device, which is in a part of memory known to all other processors, and a local server process, which handles all incoming messages. Using this method, a small portion of the Sun virtual memory can be mapped into the VME space, using Sun's Direct Virtual Memory Access (DVMA). The DVMA space lies in physical memory on the Sun, and is never swapped out, thus only a small amount of memory can be reserved as DVMA space. The **xm** devices minimum memory size is that of the largest possible message. In our current implementation, the average message is less than 32 bytes long. Only one message (a *kernel printf()* message) is relatively long at 268 bytes. A buffer of one kilobyte (four times the maximum message length and 32 times the average message length) is more than adequate, while a buffer of 4 Kbytes will prevent almost all processes from blocking on an insert into message buffer because of buffer overflow. A Sun with 4 Megabytes or more of memory can usually spare 4 Kbytes for DVMA space.

A good way to demonstrate the use of the **xm** devices at the system level is by an example. Figure 3 shows a read operation on a file or device on a remote board, which occurs transparently to the user. *Processor i* is one of the real-time general purpose processors, and *Processor j* is a different processor or the Sun host. The steps in the *read* operation are as follows: (1) the C statement *read(fd,buffer,nbytes)* is called from a user's process. *Fd* is a valid file descriptor returned from a previous *open()* call. *Buffer* is a pointer into either the processor's local memory, or a valid VME bus address. *Nbytes* is the number of bytes to read. (2) The *read()* routine first determines whether the file is local or remote, and branches accordingly. The information is available from an array of structures indexed by the file descriptor, and was initialized during the *open()* call. Assuming the operation is remote, control goes on to step (3). A message is sent to the remote board's **xm** device (4). The message consists of a header and a body. The header contains the *source process ID*, and *source board ID*, which specify the origin of the message, a *message type* which determines the contents of the body, and a set of flags, which can be used to alter the default processing methods for a particular message type. In the case of the *read()* operation, the message type is *READ*, and

the contents of the message body are the file descriptor on the remote processor, the buffer pointer in local space, and the number of bytes to read. After the message is sent, the process then blocks while waiting for the *read()* operation to complete (5).

When a message is placed into the **xm** device of processor j, a *mailbox* interrupt is generated, which wakes up the server process, which in turn begins to process the message (6). Based on the message type, the server takes appropriate action. In the case of the *read()* operation, the buffer pointer is first converted into the proper VME address, in processor j's address space. This conversion makes use of the configuration file, and solves the problem of communicating across several address spaces. The server then executes a *read()* operation, which is guaranteed to be a local operation (7). Using the converted buffer pointer, the data can be placed directly into the memory of processor i (8). After the operation is complete, a reply message is sent to acknowledge success or failure of the message (9). For a *read()*, the return value is the number of bytes read, indicating a successful operation, or -1, and the accompanying *errno*, to indicate failure. The message is placed into the **xm** device of processor i (10). Note that it is possible to send a message with the *NOREPLY* flag set, which suppresses sending the reply message. This feature is useful to allow special purpose processors, which do not have **xm** devices, to still send messages; the only difference is that no reply is received to acknowledge success or signal a failure.

The server on processor i receives the message (11). This time, the message type is *REPLY*. The default action is to wakeup the process waiting for the reply, and to pass it the return values (12). Since both processes are on the same board, local semaphores and shared memory can be used to communicate between the two processes (13). No reply is sent to a *REPLY* message (14). Finally, the read process wakes up (15), and returns the value of the *read()* operation (16).

Analyzing the efficiency of the remote *read()* operation, we note the following: The data from the *read()* is placed directly into the remote processor's memory, thus there is no need for copying from a buffer on one processor to a buffer on the other. The largest overhead thus arises from sending and receiving two messages. The next section describes the **xm** devices and their efficiency for sending and receiving messages.

5.2.1 Express Mail Devices

The **xm** devices consists of a first-in-first-out circular queue, with *xmwrite()* and *xmread()* drivers to insert messages into and remove them from the queue respectively. These drivers manipulate

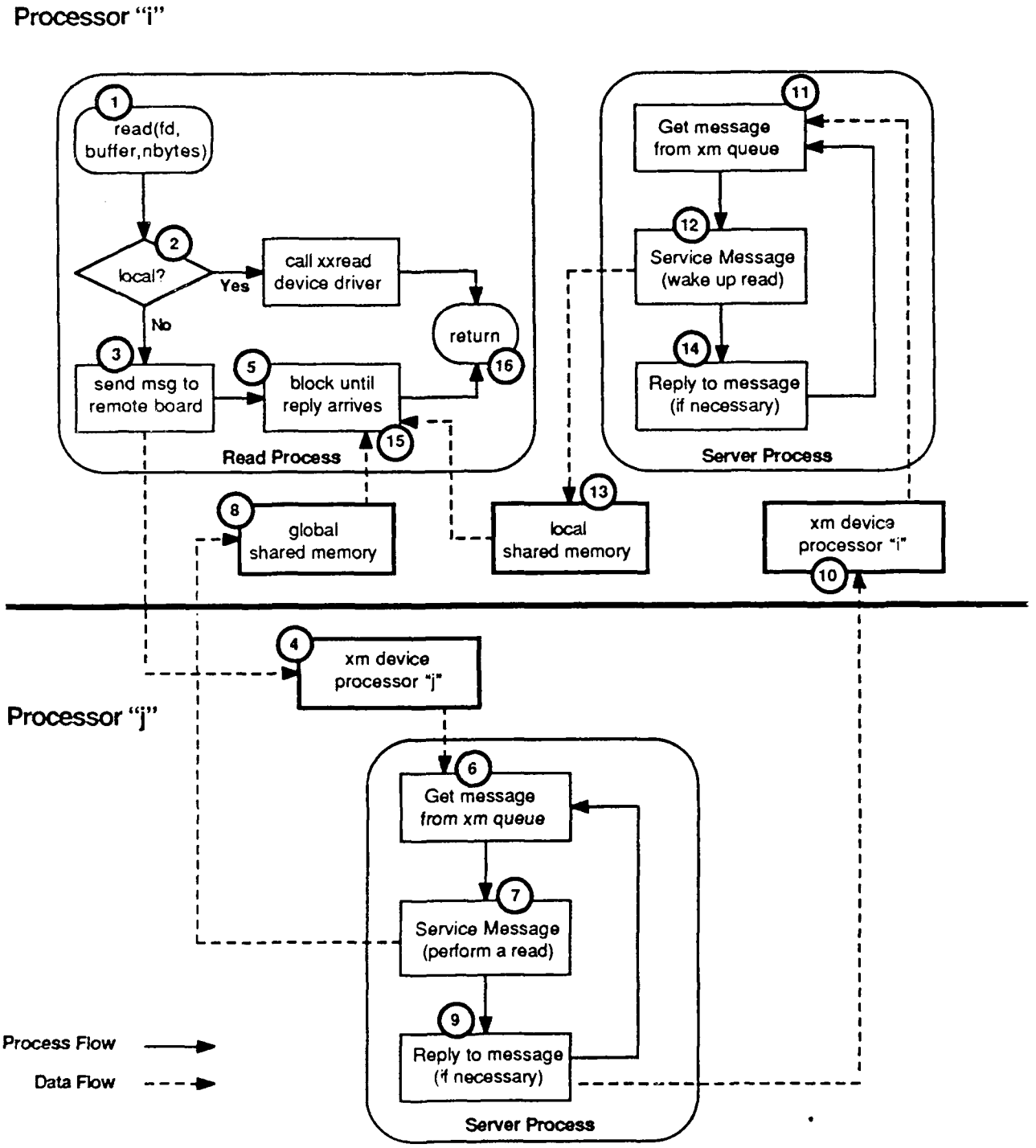

Figure 3: A remote *read()* operation

the messages using the least amount of overhead possible, while still maintaining integrity and providing mutual exclusion against being accessed by multiple processors at once. Figure 4 shows the pseudo-code for the *xmwrite()* and *xmread()* drivers. These drivers assume that the proper initialization has been performed previously.

In *xmwrite()*, checking for a legal board ID and putting the message into a packet requires only a few simple instructions. The next step is to ensure mutual exclusion when accessing the queue of the **xm** device. A *test-and-set (tas)* instruction on a mutual exclusion bit is used to obtain access. If the bit is not previously set, then the *tas()* instruction sets the bit, and proceeds to the next instruction. If it fails, however, the process does not block; rather, it will initiate a context switch so that the next ready process will run, and places itself back on the ready queue. At a later time, when it is the processes' turn to run again, it will again try the *tas()* instruction, and repeat the above steps. The reasons that encourage this type of *context-switch polling* implementation include the following:

- blocking the process trying to gain mutual exclusion would require some sophisticated synchronization to wake it up, because the process holding the mutual exclusion bit may be on a different processor.
- pure polling would waste valuable CPU time. The overhead of performing a context switch is only a fraction of the time used as compared to polling for an entire time quanta.
- The mutual exclusion bit is locked for only a very short amount of time, since the messages to be transferred are relatively short. More often than not, the *tas()* instruction succeeds on the first try, but if it does not, the lock will usually be cleared by the time the process tries again.
- A timeout mechanism prevents a process from waiting indefinitely for the lock, which can happen if the remote **xm** device dies, such as all processes terminating on the remote board terminate.
- When success is achieved the first time, the overhead for gaining mutual exclusion is limited to one *tas()* instruction and one comparison.

After mutual exclusion is obtained, a check is made to see if there is enough room in the buffer. If not, *context-switch polling* is again used until there is enough room. However, since the average message size is less than 32 bytes, and by default the **xm** device has 4 Kbytes; 128 messages can

fit in the queue before running out of place, thus buffer overflow is rare. Copying the message to the queue is done using an optimize *bcopy()* (block copy) routine.

Usually only one server process per processor reads incoming messages, using *xmread()*, therefore the *tas()* instruction usually succeeds on first try. If it does not succeed, then the server blocks. Unlike in *xmwrite()*, only onboard processes can read messages from the **xm** device, thus a process can actually block, and wakeup via a signal from another onboard process, instead of having to use context-switch polling. The server also blocks if the buffer is empty. The server wakes up upon reception of a mailbox interrupt, which signals the arrival of a message.

Messages can be sent and received within a few microseconds. The major overhead with message passing occurs when a receiving process is not the executing process, in which case at least one context switch must occur prior to reading the message. This overhead, however, is inevitable in any multitasking system.

Xm devices are used not only for implementing remote operations, as was shown in Figure 3, but are also used as a basis in setting up global shared memory segments and message queues, implementing remote synchronization, and for communicating with the remote consoles. The next section describes the user-level communication, and their implementation using the **xm** devices.

5.3 User-level Communication

CHIMERA II provides the user-level communication package which allows user programs on different CPUs to communicate efficiently and in real-time. The facilities include interprocessor shared memory, which allows a processor to access another processor's memory directly; message passing, with options for priority and real-time handling; and semaphores, with special primitives for synchronizing processors. The *Express Mail* communication devices are the underlying mechanism for implementing all of these facilities. All of the routines described in the next few sections are available for both processes running on the real-time processors, and for non-real-time processes running on the Sun host. This feature allows user applications running on the Sun host to communicate *directly* with the real-time processors.

```
xmwrite(board_id,message);
/* Put 'message' into xm queue on board with id 'board_id' */
{
    Return Error if illegal board_id
    Put message into a packet
    timeout = 0;
    while ( tas(insert_mutex[board_id]) != OK) {
        if (timeout++ > MAXTIMES) return(Timeout Error)
        context_switch;
    }
    timeout = 0;
    while (not enough place in queue) {
        if (timeout++ > MAXTIMES) return(Timeout Error)
        context_switch;
    }
    copy message to queue of board_id
    adjust pointers to queue
    release (insert_mutex[board_id])
    unblock(empty)        /* wake up anyone waiting for the message */
}

xmread(message);
/* Get message from xm device on my board */
{
    Return Error if illegal board_id
    timeout = 0;
    while (tas(remove_mutex) != OK)
        block(mutex);     /* block waiting for mutual exclusion */
    }
    remote_mutex = 1
    while (buffer empty) {
        block(empty);     /* block waiting for a message */
    }
    get message in queue
    adjust pointers to queue
    unblock(mutex);       /* wake up anyone waiting for mutual exclusion */
}
```

Figure 4: *Read()* and *write()* drivers for xm devices

5.3.1 Shared Memory

Four routines are available for using the CHIMERA II interprocessor shared memory facility: *shmCreate()*, *shmDetach()*, *shmAttach()*, and *shmDestroy()*.

ShmCreate(board,segment,size) creates a shared memory segment on the specified board, with the specified memory size. *Segment* is a symbolic name which is used by all other processes wanting to use the same segment. A pointer to the newly created segment is returned. Only one process creates a segment, while all other processes *attach* to it, using the routine *shmAttach(board,segment)*, which also returns a pointer to the memory segment. When a process is finished with a segment, it can issue the *shmDetach()* routine. The last process to finish using the shared memory issues a *shmDestroy()* command to free the memory used by the segment.

The *express mail* server for each processor is responsible for handling all requests for shared memory segments within its local address space. It also performs the necessary address space conversions, so that pointers returned can be used directly by user programs. Figure 5 gives an example of code which uses interprocessor shared memory. Figure 6 shows the process and data flow of the shared memory routines, as they are implemented using the *express mail* facility.

As a first step, process A on processor i issues the command *shmCreate("i","seg",nbytes)*, which requests that a shared memory segment of *nbytes* long be created on processor i (1). An appropriate message is placed in processor i's **xm** device (2). The server receives the message (3), and proceeds to create a shared memory segment (4,5). A pointer to "*seg*" is then returned to the calling process (6). Note that onboard processes communicate via local shared memory (7). Process A can then resume and use the shared memory segment at will (8).

Meanwhile, a second process B on processor j also wants to use the segment, so the *shmAttach()* command is issued (9). The message is placed in the **xm** device of processor i, since the shared memory segment lies on board i (10). The server processes the attach in the same manner (11,12,13). If necessary, the server converts the pointer into the proper address space so that process B can use it directly. It then places the pointer into the **xm** device of processor j (14). Processor j's server gets the message, and passes the pointer to process B (15,16,17,18). Process B can then also use the shared memory segment (19). Note that for simplicity, the example does not show any form of mutual exclusion when accessing the shared memory segment. If needed, the remote semaphores described in Section 5.3.3 can be used. When processor B is finished, it calls *shmDetach()* (20),

```
typedef struct {        /* The data structure stored in */
    int x;              /* the shared memory segment */
    float y;
} shmData;

/* The following code runs on processor "i".  */
/* Note that for simplicity, the code ignores */
/* potential problems with mutual exclusion.  */

processA_main()
{
    shmData *sd;

    sd = (shmData *) shmCreate("i","seg",sizeof(shmData));

    sd->x = 10;         /* use the shared memory */
    sd->y = 4.5;
    /* Could do lots more stuff */

    /* Keep trying to destroy until successful */
    while (shmDestroy("i","seg") == -1) ;
}

/* The following code runs on processor "j" */

processB_main()
{
    shmData *sd;

    sd = (shmData *) shmAttach("i","seg");

    printf("x = %d, y = %f\n", sd->x,sd->y);
    /* Could do lots more stuff */

    shmDetach("i","seg");       /* Detach from shared memory */
}
```

Figure 5: Example of Interprocessor Shared Memory

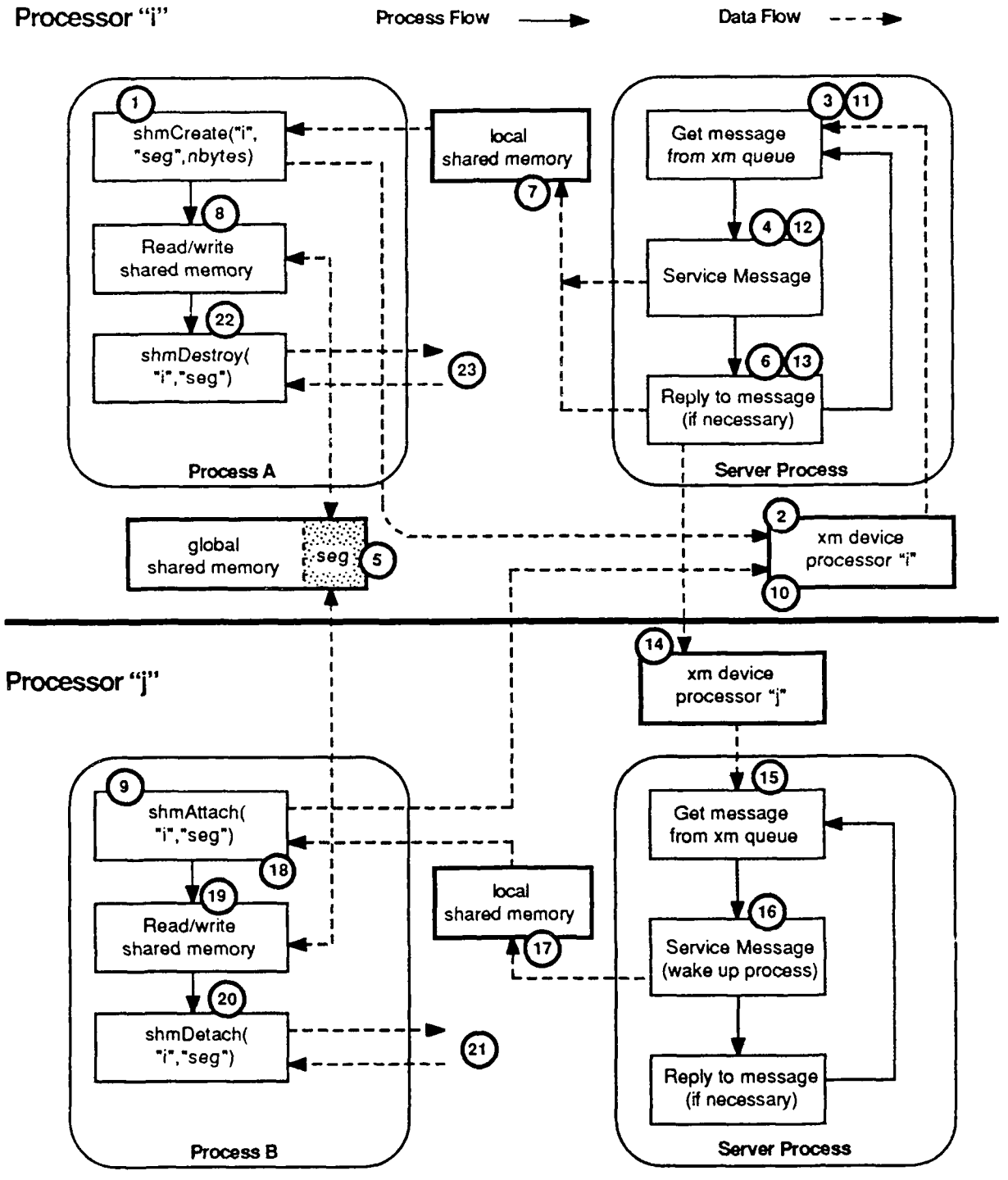

Figure 6: Implementation of Interprocessor Shared Memory

which again sends a message to the server on processor i, and wait for a reply indicating successful detachment (21). Process A can free up the memory by issuing the *shmDestroy()* command (22,23). Note that if not all processes have detached from the shared memory, *shmDestroy()* returns an error.

The advantage of this implementation of shared memory is that segments can be referred to by symbolic names. Once attached to a segment, processes can use the shared memory segment just as though the memory was on board, in the most efficient manner possible.

5.3.2 Message Passing

The message passing system is implemented much in the same way as the shared memory, using the routines *msgCreate(), msgAttach(), msgDetach(),* and *msgDestroy()* to control access to message queues, and the routines *msgSend()* and *msgReceive()* for sending and receiving typed messages.

As with the shared memory, the board name and queue name only have to be specified for *msgCreate()* and *msgAttach()*. These routines return an identifier, which is used in all subsequent operations. A process can thus send and receive messages transparently between processors.

The message passing system gives the option of specifying the queuing system to be used. It can be either priority-based, deadline-time based, or first-in-first-out. All messages are typed. A process receiving a message can also specify the type of message to be received, ignoring all other messages in the queue. *MsgReceive()* also offers the option of *blocking, non-blocking,* or *polling* retrieval of messages. If the default *blocking* mode is used, then the process waits until a message arrives before returning. If the *non-blocking* mode is selected, *msgReceive()* returns an error code if no messages are in the queue. The *polling* mode allows highly synchronized processes to receive and process messages as quickly as possible. By far, the major overhead occurring in any part of the communication package offered by CHIMERA II is the time spent by a process, which is blocked waiting for a message, to context switch back into the CPU. Using the *polling* mechanism, a process that expects a message to be arriving, can arrange to wakeup before the message actually arrives, and wait for the message to arrive by polling the queue, thus receiving a message only microseconds after it is sent. Timer interrupts can force the process to swap back out; however, the *splx()* primitive provided by the kernel can force a process to poll indefinitely until a message arrives.

5.3.3 Semaphores and Synchronization

The semaphore mechanism in CHIMERA II is consistent with the shared memory and message passing facilities, offering the routines *semCreate()*, *semAttach()*, *semDetach()*, and *semDestroy* to control access to semaphores, and provide transparent access to semaphores in subsequent calls. The routines *semP()* and *semV()* are used to perform the standard semaphore operations *P()* and *V()* remotely.

Two additional routines, *syncWait()* and *syncSignal()* are used for obtaining accurate synchronization among processors. The *syncWait()* locks the process in the CPU, and polls for an incoming synchronization signal. Processing then resumes immediately upon reception of the signal. Any other processor, or the user from the terminal interface, can issue the synchronization signal by calling *syncSignal()*, which sends a signal to every processor waiting for it. This mechanism allows two or more processors to be synchronized within microseconds of each other. A typical use for this feature is for the user to download and start executing a program on each CPU. Each program performs all initialization, then call *syncWait()*. Once all boards have been downloaded and initialized, and the user is ready to start the system, the *syncSignal()* can be issued, guaranteeing that all processors start within a few microseconds of each other.

6 Summary

CHIMERA II has been designed with the goal of supporting real-time sensor-based control applications. It is a multiprocessor and multitasking UNIX-like environment. Among the many features, it provides low-overhead interprocessor communication, in the forms of shared memory, message passing, and remote semaphores. A high-performance kernel, which supports a variety of real-time schedulers and low context switching times, allows CHIMERA II to be used with the most demanding of control algorithms. The flexibility of CHIMERA II allows the user to fine-tune the operating system to the application's needs, by providing simple software and hardware interfaces to support all types of sensor-based control applications.

7 Acknowledgements

The research reported in this paper is supported, in part, by U.S. Army AMCOM and DARPA under contract DAAA-2189-C-0001, NASA under contract NAG5-1091, the Department of Electrical and Computer Engineering, and The Robotics Institute at Carnegie Mellon University. Partial support for David B. Stewart is provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) through a Graduate Scholarship.

References

[1] J. F. Ready, "VRTX: A real-time operating system for embedded microprocessor applications," *IEEE Micro*, vol. 6, pp. 8–17, August 1986.

[2] L. Kirby, "Real-time software controls mars rover robot," *Computer Design*, vol. 27, pp. 60–62, November 1 1988.

[3] D. E. Schmitz, P. Khosla, R. Hoffman, and T. Kanade, "CHIMERA: A real-time programming environment for manipulator control," in *1989 IEEE International Conference on Robotics and Automation*, (Phoeniz, Arizona), May 1989, pp. 846–852.

[4] D. E. Schmitz, P. K. Khosla, and T. Kanade, "The CMU Reconfigurable Modular Manipulator System," in *Proceedings of 18-th ISIR*, (Australia), ISIR, 1988.

[5] T. Kanade, P. K. Khosla, and N. Tanaka, "Real-time control of the CMU Direct Drive Arm II using customized inverse dynamics," in M. P. Polis, ed., (*Proceedings of the 23rd IEEE Conference on Decision and Control*), (Las Vegas, NV), December 12-14, 1984, pp. 1345–1352.

[6] R. F. Rashid, "Threads of a new system [MACH]," *UNIX Review*, vol. 4, pp. 37–49, August 1986.

[7] W. Zhao, K. Ramamritham, and J. A. Stankovic, "Preemptive scheduling under time and resource constraints," *IEEE Transactions on Computers*, vol. C-36, pp. 949–960, August 1987.

[8] H. Tokuda, J. W. Wendorf, and H.-Y. Wang, "Implementation of a time-driven scheduler for real-time operating systems," in *Proc. IEEE Real-Time Systems Symposium*, December 1987.

[9] G. C. Kessler, "Inside TCP-IP (Transmission Control Protocol-Internet Protocol)," *Lan Magazine*, pp. 134–142, July 1989.

[10] Sun Microsystems, Inc., *Network Programming*, 1987.